

# Pitfalls of JESS for Dynamic Systems

Rajkumar Thirumalainambi  
PSGS @ NASA Ames Research Center  
Mail Stop 269-2 Moffett Field, California 94035 USA  
rajkumar@mail.arc.nasa.gov

## Abstract

*We considered different varieties of inference engines for a sub-system of Mission Control Technologies (MCT) being developed at NASA Ames Research center. One inference engine, Jess, is attractive due to its benchmark results, Java API, and being a stable software product. The outstanding issues of Jess with respect to MCT are the way its Java Bean defines 'defclass' and slots, and converting between or mapping to MCT interfaces. The slot names and 'def-class' are static in Jess rules, and at run time in a given knowledgebase it cannot be modified. Distributing Jess to outside federal agencies and licensing is another issue.*

## 1. Introduction

In modeling systems, knowledge can be represented in many ways. Each system typically has a distinction between data and rules. Data is information written in one language (sometimes very simply without negation like in a basic Resource Description Framework (RDF) [3]). The rules control the inference steps which the inference engine makes. The rules are written in a restricted language so as to preserve the computability property with validation of facts. A real inference system can work backwards or forwards (or both). We looked at 20 different inference engines from academic to industrial types based on operating system, application programming interfaces (APIs) that support specific languages and dependencies, and supporting ontology based systems. A robust inference engine should support general purpose as well semantic based systems, and specifically for MCT requirements, inference engines should support RDF and web ontology language (OWL) semantic languages through suitable translators.

The MCT Composition engine uses an inference engine, rule base, rule constructor, translator, inference engine selector and an adapter to applications. In this paper, we discuss the inference engines and rule formats supported by the inference engine. The inference engines speed is a ma-

jor factor during deployment in real time missions. The inference engines can be invoked locally or remotely from a central server with local or server based knowledgebases. The MCT's core inference engine provides a declarative programming language and is flexible enough to match the semantics of MCT application and ontology models using suitable interfaces. Ontologies store the concepts used to describe the application information. Overall MCT leverages many emerging technologies for constructing a unique plug-n-play component architecture. The MCT composition engine with suitable inference engines provides a programmatic environment for RDF, OWL and evolving ontology languages using a rule based inference engine. In section 2, I discuss the overall requirements of MCT composition engine. Section 3 overviews Jess capabilities, and section 4 discusses the integration of Jess with MCT.

## 2. Mission Control Technology (MCT) Requirements

The Mission Control Technologies project is developing a component based reusable architecture which can be applied for multiple missions across NASA. The core requirements of MCT are:

- Reuse of components in many contexts
- Dynamic addition of attributes and behaviors to components
- Autonomous state change notification of components
- API domain independence
- Lifecycle management

Based on the requirements, MCT started developing a component based infrastructure with suitable subsystems. The details of MCT are given in reference [5]. One MCT subsystem is the composition engine with its multiple inference engines for decision making in composing the User

Interface (UI) and other processes. MCT minimum requirements of inference engines, rules expressions based on ontology, translators, and necessary APIs to connect applications are as follows:

- Support forward chaining, backward chaining inference based on RETE algorithm [6]
- Frame-based approach to represent knowledge
- Support RDF, OWL, DARPA agent markup language (DAML) with logical validation
- Support server and client based inference
- Multi-ontology reasoning with a way of combining knowledgebases
- Handle high volumes of rules
- Offer a collaborative environment for model review and refinement of rules dynamically
- Reconfigure rule engine dynamically (user can select any inference engine)
- Manage dynamic changes in rules autonomously in working memory
- Versioning of rules and conflict resolution of policies (rules)
- User interfaces for validation of rules, editing of rules, querying rules (Tools)
- Statistics of execution of inference which characterize the health of inference engine (Tools)
- Advanced search capability with suitable filters in rule base (Query Tools)
- Runtime management of inference engine and reasoning mechanism

Based on the above requirements, the initial criteria for selecting an inference engine is based on performance (standard bench mark test). In Table 1, the comparative results are given for the Manners 128 standard benchmark test performed using ILOG Jrules 5.01, OPSJ 6.0, and Jess 6.1, (an inference engine from Sandia National Laboratory [1]). In figure 1, a different benchmark test between Jess, Microsoft Business Rule Engine, and Drools using 10000 rules [7] and time in milliseconds to execute the rules are shown [8]. All tests were conducted with Microsoft windows XP on the same machine (w.r.t Figure 1). Based on the initial performance results, Jess and Drools are promising inference engines considered for MCT composition engine. According to table 1, Jess did worse than Jrules w/hash. Jrules is

a commercial product whereas Jess can be used for federal research and mission purposes without any cost.

In comparing features, Drools uses a natural language based rule set, editing, and development (drl and dsl files), whereas Jess requires specific syntax to represent rules (JessML or clips) [4]. Drools and Jess support facts which can be expressed in standard Java class and methods. Jess and Drools follow the Java Bean approach to insert Java objects in working memory. Java Beans properties are similar to the list of slots in the facts of Jess and Drools. A Java Bean property is a pair of methods that express in a standard way String name 'value' via String getValue() and void setValue(String str). The get method is used to read the value of the property, while the set method changes the variable to a specific value.

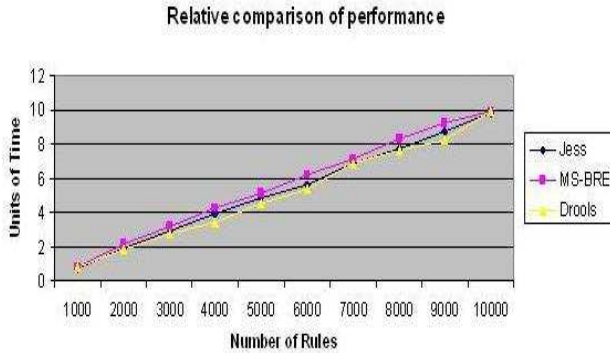
**Table 1. Benchmark results of JRULES (ILOG) and JESS**

Benchmark	Manners 128 standard benchmark test				
Platform	Windows XP	Mac 1GHz G4	Dual SPARC	Dual . Win Serever	2 x Mac G5
CPU Speed (GHz)	1.9	1.0	1.0	1.0	2.0
JRULES 5.0.1 no hash (milliseconds)	89.00	131.00	73.67	73.68	56.80
JRULES 5.0.1 with hash	3.70	8.70	4.66	9.60	3.45
OPJS 6.0	7.93	18.35	9.04	19.58	na5
JESS 6.1 p6, p7	66.50	164.00	73.20	na	na

The java.beans API includes a class named Introspector that can check Java Beans and find properties defined according to get/set naming convention. Jess and Drools use Introspector to generate deftemplate which can serve as facts. Since Jess and Drools are well supported and stable products, we progressed to implement Jess as an initial inference engine for MCT composition engine. The specific capabilities of the Jess inference engine are discussed in the next section.

### 3. JESS Capabilities

Jess offers a scripting language to implement a knowledge base which is very similar to C Language Integrated Production System (CLIPS) [2]. It uses first order logic and supports both forward and backward chaining. Backward chaining lends the system a "goal seeking" behavior,



**Figure 1. Benchmark results of JESS and DROOLS.**

whereas forward chaining looks for satisfied rules on its own after some facts are known. It can be used in a multithreaded environment and uses the RETE algorithm for optimized speed. The `Jess.Rete` class synchronizes internally in many ways for efficiency. It can directly manipulate and reason about Java objects. It offers a built-in translator to translate a CLIPS format knowledge base to the Jess ML format, which resembles RuleML and XML formats. It offers to divide and conquer a large knowledge base using a `defmodule` concept. The `defmodule` allows managing a knowledge base in such way that it can access a specific knowledge base using focusing individual modules.

Jess's working memory is stored in a complex data structure with multiple indexes, so that searching the working memory is extremely fast. There are two ways in which facts can be represented in working memory: (i) Unordered facts (ii) Ordered facts. An unordered fact is like a row in a relational database table. The assertion of a slot can be in any order and can be used to represent any general scenario. An ordered fact lacks the structure of named fields; in short, it is a flat list. A form of unordered facts is called shadow facts which represent Java objects. The working memory representation of a Java Bean can be either static or dynamic. In dynamic cases, the Java Bean property listener is implemented to note any changes for that specific Java object. Shadow facts always have slot values for "class", which is the instance of `java.lang.Class` representing the class of the backed Java object, and "OBJECT", which is a reference to the backed Java object itself. In MCT composition engine, shadow facts will be used with the 'IComponent' interface in a dynamic way. The IComponent is the central interface for MCT. Jess supports interfaces and classes to represent facts.

Jess offers conditional logic checking on the left hand side of a rule. It allows multiple nesting of boolean and conditional logic with salient features to execute rules. De-

fquery is a special kind of rule with no right hand side. The query provides matching facts in the form of an iterator. Jess provides a user function interface and user package to define user defined functions. The user package can have multiple user defined functions. Most of the Jess API is designed as user functions and user packages. These functions allow the user to add more application oriented functions. The user can use the functions in rules. Jess can be used in a RMI based rule server by creating suitable stubs and a skeleton. Drools is also similar to Jess in many ways. With respect to drools, facts are objects (Java beans) from the application that are being asserted into the working memory. Strings and other classes without getters and setters are not valid facts and can't be used with Field Constraints which rely on the Java Bean standard of getters and setters to interact with the object.

#### 4. Integrating JESS with MCT

In MCT, many components are developed to support any mission. The components are represented by representation and suitable role. Depending on the role, the composition can take place. The composition policies are the knowledge base, which has been constructed using the IComponent interface. MCT does not follow the strict Java Bean standards for get and put (instead of set). In the IComponent interface, getter methods have arguments which enable dynamic capability, whereas in the standard Java bean approach, getter methods have null arguments. An example of the getter and setter methods of Icomponent is shown below:

```
public List getValueNotes(String fieldName, Object
fieldValue, String noteName);
public void putFacetValues(String fieldName, String
facetName, List facetValues);
```

Since MCT has many components with multiple roles, getter methods need arguments to access a specific component within a dynamic environment. The behavior of components is controlled by the IActor interface. Since MCT does not strictly adhere to Java Bean standards, each method has to be wrapped into a specific Java bean method to access the facts and slots. Jess can not construct rules based on MCT classes directly but rather through MCT wrapper classes which leads to the slower performance. The number of rules in the knowledge base will exponentially increase due to the roles of components in MCT. In MCT, the names of the fields of a Java class are runtime variables which contribute its dynamism. Unfortunately, the Jess template is like a Java class; its slot names are fixed when it is defined and cannot be changed.

Since MCT provides the infrastructure, a user can define new field names for specific applications and dynamically, but it does not get reflected in the Jess knowledge base. As long as rules are static and the domain is static, the

Jess knowledge base can be constructed initially and can serve the purpose of inference. In dynamic environments like MCT, the rules will not change autonomously or cannot be constructed. This can be done by a learning mechanism of all components interactions over a period of time, but this option may not be very efficient with respect to MCT. The integration of Jess to do dynamic composition and dynamic rule generation needs more work, and it will probably require an architectural change in Jess and its parser.

## 5 JESS Parser

The Jess parser strictly follows the Jess scripting language syntax which is defined in CLIPS syntax. Any conversion of the Jess rule format requires a significant amount of effort in terms of computing power. At run time specific Jess knowledge base can not be modified or changed during fact matching. Salient values can be integers, global variables, or function calls. In the case of a tie in the salient values of rules, two different conflict resolution strategies are currently available (CLIPS has seven): depth (LIFO) and breadth (FIFO). In either case, if several rules are activated simultaneously (i.e. by the same fact assertion event) the order in which these several rules fire is unspecified. The user has to implement a specific strategy to address salient features, when several rules fire. In the `Jess.Rete` class, the `reset` function will not remove shadow facts from working memory. Shadow facts must be retracted explicitly to reset the engine with new facts. Shadow facts cannot be asserted. Instead, the backed Java object must first be constructed and then added to working memory as a fact using `'definstance'`. Jess inherits lot of Java, but it is not an object oriented system. Traditional expert systems work by matching patterns, where rules are static. Converting traditional inference engines to adapt to dynamic MCT requirements needs more investigation.

## 6 Conclusion

MCT requirement for an inference engine should handle dynamism in rules and variables and dynamic facts handling. The reason for selecting Jess was due to bench mark results and stable Java API. The key issue regarding its static nature of rules and Java Bean approach of inserting facts to working memory does not meet MCT requirements to operate in a dynamic environment. A dedicated inference engine is planned to be constructed for MCT to handle dynamism and the research is underway.

## Acknowledgement

The author would like to express sincere thanks to Mr. Jay Trimble, Code TI for funding and necessary support to

carry out this research. The author would like to acknowledge useful comments, suggestions and discussions from Dr. Nikunj Oza and Mr. Francis Enomoto NASA Ames Research Center.

## References

- [1] <http://www.kbsc.com/Performance2000-2005.xls>.
- [2] Clips reference manual volume i, basic programming guide. <http://www.ghg.net/clips/download/documentation/bpg.pdf> version 6.23, January 2005.
- [3] Rdf. <http://www.w3.org/RDF>.
- [4] F. E.J. Jess, the rule engine for java platform. <http://herzberg.ca.sandia.gov/jess/docs/70,version7.0a5,DraftFebruary,2005>.
- [5] S. Fonseca. Engineering degrees of agency. In *Fifth International Workshop on Software Engineering for large scale multi-agent systems (SELMAS)*, Shanghai, China, May 2006.
- [6] R. Forgy C. L. A fast algorithm for the many pattern/many object pattern match. *Artificial Intelligence*, 19(1982), 1991.
- [7] M. Proctor. Drools-jboss rules. <http://labs.jboss.com/portal/jbossrules/docs>.
- [8] C. Young. Microsoft's rule engine scalability results - a comparison with jess and drools, 2004. <http://geekswithblogs.net/cyoung/articles/54022.aspx>.